**ESERCIZIO pthread-1**

In un sistema operativo che realizza i thread a livello kernel, i thread T11 e T12 del processo P cooperano scambiando messaggi attraverso un buffer circolare di n celle (numerate da 0 a n-1), ciascuna capace di contenere un messaggio.

Un possible schema di cooperazione, realizzato con i semafori CelleLibere (valore iniziale n) e MessaggiGiacenti (valore iniziale 0) e con le variabili IndDeposito e IndPrelievo (interi nell’intervallo [0,n); valori iniziali 0) è il seguente:

Thread T11

…………

ProduceMessaggio(mesaggio);

**wait**(CelleLibere);

DepositaNelBuffer(messaggio, IndDeposito);

IndDeposito= IndDeposito + 1 %n;

**signal**(MessaggiGiacenti);

………

Thread T12

…………

**wait**(MessaggiGiacenti);

PrelevaDalBuffer(mess, IndPrelievo);

IndPrelievo= IndPrelievo + 1 %n;

**signal**(CelleLibere);

ConsumaMessaggio(mess);

………

Trasformare il precedente scherma usando i meccanismi mutex e condition della libreria p\_thread (con operazioni p\_thread\_mutex\_lock, p\_thread\_mutex\_unlock, p\_thread\_condition\_wait e p\_thread\_condition\_signal) utilizzando:

- per il primo meccanismo la variabile MutexBuffer (di tipo pthread\_mutex)

- per il secondo le variabili AttesaDeposito e AttesaPrelievo di tipo pthread\_condition

- le variabili condivise CelleLibere (valore iniziale n) e, analogamente allo schema precedente, le variabili IndDeposito e IndPrelievo (interi nell’intervallo [0,n))).

**SOLUZIONE**

Thread T11

…………

ProduceMessaggio(messaggio);

***p\_thread\_mutex\_lock*** *(MutexBuffer);*

**while** CelleLibere== 0 ***p\_thread\_condition\_wait***(AttesaDeposito, MutexBuffer);

DepositaNelBuffer(messaggio, IndDeposito);

CelleLibere -- ;

IndDeposito= IndDeposito + 1 %n;

***p\_thread\_condition\_signal***(AttesaPrelievo);

***p\_thread\_mutex\_unlock****(MutexBuffer);*

………

Thread T12

…………

***p\_thread\_mutex\_lock*** *(MutexBuffer);*

**while** CelleLibere== n ***p\_thread\_condition\_wait***(AttesaPrelievo, MutexBuffer);

PrelevaDalBuffer(mess, IndPrelievo);

CelleLibere ++ ;

IndPrelievo= IndPrelievo + 1 %n;

***p\_thread\_condition\_signal***(AttesaDeposito);

***p\_thread\_mutex\_unlock*** *(MutexBuffer);*

ConsumaMessaggio(mess);

…

**ESERCIZIO pthread-2**

Un parcheggio per auto della capacità di 10 posti è dotato di un unico ingresso I e un’unica uscita U. L’ingresso e l’uscita sono controllate da sbarre. Le auto sono thread di uno stesso processo, conformi allo standard POSIX, che si sincronizzano mediante il meccanismo *mutex* (sul quale sono definite le operazioni pthread\_mutex\_lock e pthread\_mutex\_unlock), e il meccanismo *condition* (sul quale sono definite le operazioni pthread\_cond\_wait e pthread\_cond\_signal).

Per l’utilizzo del parcheggio, i thread condividono la variabile *PostiDisponibili,*  con valore iniziale 10, la variabile *MutexParcheggio* di tipo *mutex*, e la variabile *PostoLibero,* di tipo *condition.*

Si chiede di risolvere il problema completando lo pseudo codice sotto riportato con le opportune operazioni di sincronizzazione.

**SOLUZIONE**

<l’auto arriva all’ingresso I>

**pthread\_mutex\_lock**(&MutexParcheggio)

**while** PostiDisponibili == 0 **pthread\_cond\_wait**(&PostoLibero,&MutexParcheggio);

*//l’auto attende la disponibilità di un posto//*

<la sbarra si alza e l’auto entra nel parcheggio>

<l’auto sceglie un posto libero e lo occupa >

PostiDisponibili - -

**pthread\_mutex\_unlock**(&MutexParcheggio);

<auto parcheggiata>

**pthread\_mutex\_lock**(&MutexParcheggio);

<l’auto libera il posto che aveva occupato e si dirige all’uscita U> PostiDisponibili ++

**pthread\_cond\_signal**(&PostoLibero);

**pthread\_mutex\_unlock**(&MutexParcheggio);

<la sbarra si alza e l’auto esce dal parcheggio>

**ESERCIZIO pthread-3**

In una banca vi sono 10 sportelli per il servizio ai clienti, situati in una sala d’aspetto. I clienti entrano nella sala d’aspetto, attendono che vi sia uno sportello libero e, quando si verifica questo evento, lo individuano, lo raggiungono e ottengono il servizio voluto. Quindi liberano lo sportello e tornano in sala d’aspetto per uscire dalla banca. Si chiede di risolvere il problema completando lo pseudo-codice sotto riportato con le opportune operazioni di sincronizzazione, nella seguente ipotesi:

- i clienti sono thread di uno stesso processo, conformi allo standard POSIX, che si sincronizzano mediante la variabile *mux* di tipo *mutex* (sulla quale sono definite le operazioni pthread\_mutex\_lock e pthread\_mutex\_unlock), e la variabile *SportelloLibero* di tipo *condition* (sulla quale sono definite le operazioni pthread\_cond\_wait e pthread\_cond\_signal). I thread utilizzano inoltre la variabile *SportelliLiberi*, di tipo intero e inizializzata al valore 10.

**SOLUZIONE**

<il cliente entra nella sala d’aspetto>

**pthread\_mutex\_lock**(&Mutex)

**while** SportelliLiberi == 0 **pthread\_cond\_wait**(&SportelloLibero,&Mutex);

*//esiste uno sportello libero: il cliente lo individua e lo raggiunge//*

SportelliLiberi - -

**pthread\_mutex\_unlock**(&Mutex);

*<il cliente ottiene il servizio richiesto>*

**pthread\_mutex\_lock**(&Mutex);

*<il cliente lascia lo sportello e torna in sala d’aspetto>*

SportelliLiberi ++

**pthread\_cond\_signal**(&SportelloLibero);

**pthread\_mutex\_unlock**(&Mutex);

*<il cliente lascia la sala d’aspetto>*

**ESERCIZIO pthread-4 (Problema dei Lettori-Scrittori)**

Nel problema dei lettori-scrittori, *n* processi lettori e *m* processi scrittori competono per l’accesso ad una struttura dati condivisa. I processi lettori utilizzano la struttura dati esclusivamente per leggere, senza modificare i dati, mentre i processi scrittori la utilizzano senza restrizioni, con la possibilità di modificare i dati.

Per evitare interferenze, i processi scrittori accedono alla base dati in mutua esclusione (rispetto agli altri scrittori e ai lettori), mentre i processi lettori accedono alla base dati in mutua esclusione rispetto agli scrittori, ma senza vincolo di mutua esclusione rispetto agli altri lettori (in altre parole, più lettori possono accedere concorrentemente alla struttura dati).

Per la soluzione del problema, si utilizzano i seguenti dati condivisi da tutti i processi:

* *LettoriAmmessi*: intero non negativo; valore iniziale 0
* *LettoriInAttesa*: intero non negativo; valore iniziale 0
* *ScrittoreAmmesso*: intero nel intervallo [0, 1]
* *ScrittoriInAttesa*: intero non negativo; valore iniziale 0

Nell’ipotesi che i lettori e gli scrittori siano thread di uno stesso processo, si chiede di risolvere il problema utilizzando la variabili *P\_mutex,* di tipo *mutex,* per la mutua esclusione e le variabili *AttesaLettori* e *AttesaScrittori*, di tipo *condition*, rispettivamente per la sospensione dei lettori e degli scrittori, e introducendo i comandi POSIX *pthread\_mutex\_lock, pthread\_mutex\_unlock, pthread\_cond\_wait e pthread\_cond\_signal* nello schema di soluzione proposto.

**SOLUZIONE**

*{nota: questa soluzione dà precedenza ai lettori e non evita l’attesa indefinita per gli scrittori}*

**Lettore\_i**

{

………………..

*// prologo dell’accesso in lettura//*

**pthread\_mutex\_lock**(&P\_mutex);

LettoriInAttesa++;

**while** ScrittoreAmmesso<> 0 **pthread\_cond\_wait**(&AttesaLettori, &P\_mutex);

*//il lettore attende fino al rilascio della base dati da parte dello scrittore//*

LettoriInAttesa - -; LettoriAmmessi ++;

**pthread\_mutex\_unlock**(&P\_mutex);

< esegue accesso in lettura >

*// epilogo dell’accesso in lettura//*

**pthread\_mutex\_lock**(&P\_mutex);

LettoriAmmessi - -;

**if (**LettoriAmmessi== 0) **pthread\_cond\_signal**(&AttesaScrittori);

*//si riattiva, se esiste, uno scrittore in attesa di accedere//*

**pthread\_mutex\_unlock**(&P\_mutex);

………………..

}

**Scrittore\_j**

{

………….

*// prologo dell’accesso in scrittura//*

**pthread\_mutex\_lock**(&P\_mutex);

ScrittoriInAttesa ++;

**while (**LettoriAmmessi > 0 **or** ScrittoreAmmesso<> 0) **pthread\_cond\_wait**(&AttesaScrittori, &P\_mutex);

*//lo scrittore attende fino al rilascio della base dati da parte dello scrittore o dei lettori che la utiilizzano//*

ScrittoriInAttesa - -;Scrittore Ammesso= 1,

**pthread\_mutex\_unlock**(&P\_mutex);

< esegue accesso in scrittura >

*// epilogo dell’accesso in scrittura//*

**pthread\_mutex\_lock**(&P\_mutex);

ScrittoreAmmesso= 0;

**if** (LettoriInAttesa>0)

**for** (i= 0, i< LettoriInAttesa, i ++) **pthread\_cond\_ signal**(&AttesaLettori);

*//si riattivano tutti i lettori in attesa di accedere//*

**else pthread\_cond\_ signal**(&AttesaScrittori);

*//si riattiva, se esiste, uno scrittore in attesa di accedere//*

**pthread\_mutex\_unlock**(&P\_mutex);

………………..

}

**ESERCIZIO pthread-5**

Nel problema dei “filosofi a cena”, *N* filosofi, individuati da un indice i compreso nell’intervallo *[0, N)* si riuniscono per la cena in un ristorante cinese, dove occupano un tavolo circolare, apparecchiato con un piatto per ogni filosofo e un bastoncino interposto fra ogni coppia di piatti adiacenti. Per mangiare, il filosofo di indice i deve acquisire il bastoncino di indice *i* e quello di indice *(i+1) mod N*. Se il bastoncino di indice *i* è utilizzato dal filosofo di indice *(i-1) mod N*, oppure se bastoncino di indice *(i+1) mod N* è utilizzato dal filosofo di indice *(i+1) mod N*, il filosofo di indice *i* deve attendere la disponibilità di entrambi i bastoncini. Ogni filosofo alterna ciclicamente, con velocità arbitraria, tra uno stato in cui pensa, uno in cui desidera mangiare e uno in cui mangia.

Si suppone che i filosofi siano thread di uno stesso processo che condividono il vettore *stato,* dove *stato[i]* è associato al filosofo di indice *i* e può assumere i valori *“pensa”*, *“HaFame”* e *“mangia”*, con i seguenti significati:

* *stato[i]= “pensa”* significa che il filosofo di indice *i* non è in possesso di nessun bastoncino;
* *stato[i]= “HaFame”* significa che il filosofo di indice *i* richiede i due bastoncini che gli sono necessari per mangiare;
* *stato[i]= “mangia”* significa che il filosofo di indice *i* è in possesso dei due bastoncini

Si chiede di risolvere il problema utilizzando la variabile *MutexTavolo*, di tipo *mutex* e il vettore *filosofo*, dove *filosofo[i]* è una variabile di tipo *condition* associata al filosofo di indice *i*, e introducendo i comandi POSIX *pthread\_mutex\_lock, pthread\_mutex\_unlock, pthread\_cond\_wait e pthread\_cond\_signal* nello schema di soluzione proposto.

**SOLUZIONE**

**ProgrammaDelFilosofo\_i**

{

i= IndiceDelFilosofo; stato[i]= pensa;

**while** (true) {

**pthread\_mutex\_lock**(&MutexTavolo);

stato[i]= HaFame;

*//la transizione nello stato “HaFame” corrisponde alla richiesta dei due bastoncini//*

**while**  (stato[(i- 1) **mod** N] == mangia **or** stato[(i+ 1) **mod** N]== mangia)

**pthread\_cond\_wait**(&filosofo[i]),&MutexTavolo)

**//** *il filosofo si sospende nello stato “pensa”; la richiesta rimane pendente//*

stato[i]= mangia;

*// la transizione nello stato “mangia” corrisponde all’assegnazione dei due bastoncini //*

**pthread\_mutex\_unlock**(&MutexTavolo)

< il filosofo di indice i mangia >

*//il filosofo di indice i ha finito di mangiare e torna nello stato “pensa” //*

**pthread\_mutex\_lock**(&MutexTavolo)

stato[i]=pensa;

*// la transizione nello stato “pensa” corrisponde al rilascio dei due bastoncini //*

**if** stato[(i- 1) **mod** N]== HaFame **and** stato[(i- 2) **mod** N]<> mangia {

stato[[i- 1] **mod** N]= mangia;

**//***il filosofo (i-1)* ***mod*** *N ha a disposizione entrambi i bastoncini //*

**pthread\_cond\_signal**(&filosofo[(i-1) mod n];

}

**if** stato[(i+ 1) **mod** N]== HaFame **and** stato[(i+ 2) **mod** N]<> mangia {

stato[(i+ 1) **mod** N]= mangia;

**//** *il filosofo (i+1)* ***mod*** *N ha a disposizione entrambi i bastoncini //*

**pthread\_cond\_signal**(&filosofo[[i+1) mod n];

}

**pthread\_mutex\_unlock**(&MutexTavolo)

}

}

**ESERCIZIO pthread-6 (Problema del barbiere dormiglione)**

Nel problema del “barbiere dormiglione” si considera un negozio gestito da un unico barbiere, con una poltrona per il servizio dei clienti e un numero illimitato di sedie per l’attesa. Il barbiere e i clienti sono thread di uno stesso processo conformi allo standard POSIX e la poltrona è una risorsa, che può essere assegnata a un cliente per il taglio dei capelli, oppure utilizzata dal barbiere per sonnecchiare. All’apertura del negozio e quando non ci sono clienti in attesa di servizio, il barbiere occupa la poltrona, fino all’arrivo del primo cliente.

Quando entra nel negozio, il generico cliente ha il seguente comportamento:

* se il barbiere è addormentato, lo risveglia provocando il rilascio della poltrona, che occupa immediatamente;
* altrimenti (ciò avviene quando il barbiere è attivo per eseguire il taglio dei capelli ad un altro cliente) si blocca,
* accomodandosi su una delle sedie in attesa del suo turno;
* quando arriva il suo turno, è riattivato dal barbiere ed occupa la poltrona;
* dopo il taglio dei capelli, paga ed esce dal negozio.

Dopo che un cliente ha occupato la poltrona, il barbiere esegue il taglio dei capelli e al termine:

* se ci sono clienti in attesa del proprio turno, riattiva il primo;
* altrimenti occupa la poltrona e si addormenta.

Il problema viene risolto utilizzando i seguenti dati condivisi:

* *BarbiereAttivo: booleano;*
* *ClientiInAttesa: intero; valore iniziale 0;*

e inoltre le seguenti variabili:

* *Mux:, di tipo mutex, valore iniziale 1);*
* *AttesaTurno, di tipo condition;*
* *AttesaBarbiere, di tipo condition;*
* *TaglioCapelli, di tipo condition. : valore iniziale 0 (il cliente si sospende su questo semaforo all’inizio del taglio dei capelli e viene riattivato dal barbiere alla fine. L’attesa su questo semaforo implica l’assegnazione della poltrona al cliente).*

Si chiede di completare il programma del barbiere e il frammento di codice eseguito da ogni cliente che entra nel

negozio, inserendo gli opportuni meccanismi POSIX.

**SOLUZIONE**

**Barbiere**

{

**pthread\_mutex\_lock**(&Mux); BarbiereAttivo= false; **pthread\_mutex\_unlock**(&Mux);

*/inizializzazione; eseguita prima della generazione dei processi “cliente” //*

**while**(true) {

**pthread\_cond\_signal**(&AttesaTurno);

*// se vi sono clienti inattesa riattiva il primo; altrimenti non ha effetto //*

**pthread\_mutex\_lock**(&Mux);

SediaOccupata== true**;**

**while**(BarbiereAttivo== false) **pthread\_cond\_wait**(&AttesaBarbiere,&Mux);

*// si sospende rilasciando la mutua esclusione; sarà risvegliato da un cliente e riacquisirà la mutua esclusione //*

SediaOccupata== false **;**

*// un cliente e’ pronto per il taglio dei capelli: quello riattivato dal barbiere o quello che ha risvegliato il barbiere //*

**while**(SediaOccupata== false) **pthread\_cond\_wait**(&InizioTaglio,&Mux);

<esegue il taglio dei capelli>

BarbiereAttivo= false;

**pthread\_cond\_signal**(&TaglioCapelli);

**pthread\_mutex\_unlock**(&Mux);

<presenta ail conto al cliente, che libera la poltrona>

*// ripete il ciclo, attivando un cliente in attesa o addormentandosi //*

}

}

**Cliente**

{

…………………

*//Frammento di codice//*

< entra nel negozio >

**pthread\_mutex\_lock**(&Mux);

**while** (BarbiereAttivo) **pthread\_cond\_wait**(&AttesaTurno,&Mux);

BarbiereAttivo= true;

**pthread\_cond\_signal**(&AttesaBarbiere);

*// il barbiere è in attesa e viene risvegliato; il cliente che risveglia è appena entrato o è stato riattivato dal barbiere* ***//***

*// un cliente e’ pronto ad occupare la poltrona: si tratta di quello appena entrato in negozio, o di quello riattivato dal barbiere //*

SediaOccupata== true **; *//*** *occupa la poltrona //*

**while** (BarbiereAttivo) **pthread\_cond\_wait**(&TaglioCapelli,&Mux);

*// attende il taglio dei capelli rilasciando la mutua esclusione, che riacquista dopo la riattivazione //*

<paga ed esce dal negozio>

**pthread\_mutex\_unlock**(&Mux);

*// Fine del frammento di codice //*

…………………

}**ESERCIZIO pthread-7**

Un ristorante Fast-Food che somministra un unico tipo di hamburger è gestito da un inserviente e da un cuoco, che interagiscono attraverso uno scaffale, dove possono essere accumulati fino a 10 hamburger pronti per essere serviti. II cuoco prepara gli hamburger in sequenza e li depone sullo scaffale, eventualmente attendendo che ci sia un posto disponibile. Il generico cliente fa il suo ordine all’inserviente e quindi attende la consegna dell’hamburger. L’inserviente riceve in sequenza gli ordini, preleva gli hamburger dallo scaffale (eventualmente attendendo la disponibilità) e li consegna ai clienti, riattivandoli.

Il ristorante è un processo, i cui thread sono i clienti, l’inserviente e il cuoco. Lo scaffale è un buffer di 10 celle, ciascuna capace di contenere un hamburger*.*

Per l’interazione tra i thread si utilizzano le operazioni della libreria standard *p\_thread,* operando sulle variabili *MutexOrdini* e *MutexScaffale* di tipo *mutex,* tutte inizializzate al valore 1, e sulle variabili *AttesaOrdine, consegna, HamburgerNelloScaffale* e *PostoLiberoNelloScaffale* di tipo *condition,* tutte inizializzate con coda vuota.

Si utilizano inoltre le variabili intere (condivise) *OrdiniPendenti* e *HamburgerPronti,* con valore iniziale 0.

Il generico cliente esegue la seguente funzione:

thread GenericoCliente

…………

<entra nel negozio>;

**pthread\_mutex\_lock**(&MutexOrdini)**;**

OrdiniPendenti ++;

<ordina un hamburger e paga>;

**pthread\_condition\_signal** (&AttesaOrdine); *//ha effetto solo se l’inserviente è sospeso//*

**pthread\_mutex\_unlock**(&MutexOrdini);

**pthread\_mutex\_lock**(&MutexFittizio)**;**

*/l’associazione della condition a un mutex è obbligatoria, anche quando, come in questo caso, si voglia una sospensione incondizionata/*

**pthread\_condition\_wait** (&consegna, &MutexFittizio)

*//attende incondizionatamente la consegna//*

**pthread\_mutex\_unlock**(&MutexFittizio)**;**

………

dove la variabile *MutexFittizio* è introdotta unicamente per motivi sintattici (la funzione ***pthread\_condition\_wait*** ha sempre due parametri, il secondo dei quali è di tipo mutex), e non viene utilizzata altrove.

Si chiede di completare le funzioni eseguite dal thread *Inserviente* e dal thread *Cuoco,* inserendo negli schemi sotto riportati le operazioni sulle variabili di tipo *mutex* e sulle variabili di tipo *condition*.

**SOLUZIONE**

**thread Inserviente**

**do while** true {

**pthread\_mutex\_lock**(&MutexOrdini)**;**

**while** (OrdiniPendenti== 0) **pthread\_condition\_wait** (&AttesaOrdine, &MutexOrdini);

*//attende un ordine//*

OrdiniPendenti --;

*//gestisce gli ordini com politica FIFO//;*

**pthread\_mutex\_unlock**(&MutexOrdini);

**pthread\_mutex\_lock**(&MutexScaffale);

**while** (HamburgerPronti== 0) **pthread\_condition\_wait** (&HamburgerNelloScaffale, &MutexScaffale)

*//attende la disponibilità di almeno un hamburger nello scaffale//*

<preleva un hamburger dallo Scaffale>

HamburgerPronti --;

**pthread\_condition\_signal** (&PostoLiberoNelloScaffale);

**pthread\_mutex\_unlock**(&MutexScaffale)**;**

**pthread\_condition\_signal** (&consegna)

//consegna un hamburger, riattivando i clienti in ordine FIFO//

}

**thread Cuoco**

**do while** true {

<prepara un hamburger>

**pthread\_mutex\_lock**(&MutexScaffale);

**while** (HamburgerPronti== 10) **pthread\_condition\_wait** (&PostoLiberoNelloScaffale, &MutexScaffale);

*//attende la disponibiltà di almeno un posto libero nello scaffale //*

<deposita un hamburger sullo Scaffale>

HamburgerPronti ++;

**pthread\_condition\_signal** (&HamburgerNelloScaffale);

*//ha effetto solo se l’inserviente è sospeso//*

**pthread\_mutex\_unlock**(&MutexScaffale);

}